1 Kjøretid

Read the following code and analyze the runtime. The variable `list` can either be an `ArrayList` or a `LinkedList`, will it matter which type of list it is? Explain your answer and write down the runtime for both type of lists. Express the runtime using Big-O notation as a function of `n = list.size()`. Explain how you found the running time.

```java
for(int i = 0; i < list.size(); i++) {
    for(int j = 0; j < i; j++) {
        if (list.get(i).equals(list.get(j))) {
            return list.get(i);
        }
    }
}
```

Svar. The two for loops will have an average runtime of $O\left(\frac{1}{2}n^2\right)$. This is because the inner for loop will only iterate up to the value `i`. This means that in total, the inner for loop will iterate the following amount of times

$$\sum_{i=0}^{n-1} \frac{a_i}{n} = \frac{1}{2}(n-1)$$

With this in place, we can look at the contents of the for loop. We have the following

Operation	`LinkedList`	`ArrayList`
`list.get`	$O(n)$	$O(1)$
`list.equals`	$O(1)$	$O(1)$

As we can see, two `list.get()` operations are performed for each iteration. Another `list.get()` operation is performed if the if statement evaluates to `true`. This gives the worst-case runtime for the contents of the contents of the for loop $O(3n)$ and $O(3)$ for a `LinkedList` and `ArrayList`, respectively. In total, we end up wit the total runtimes being $O(\frac{3}{2}n^3)$ and $O(\frac{1}{2}n^2+3)$ for a `LinkedList` and `ArrayList`, respectively. Removing the constants and non-dominant terms, we obtain the following runtimes

$$\texttt{LinkedList} : O(n^3)$$
$$\texttt{ArrayList} : O(n^2)$$

2 Binary Search

Explain what Binary Search is

1. When can we use binary search and what can we use it for?
2. Explain what the runtiem of a binary search is and how we compute the runtime.

Runtime shall be expressed in Big-O notation using the paramater `n`, which is the length of the input.

Svar. Binary can be used to find an element, or the index of an element within a list. The prerequisite for using a binary search is that the list is sorted.

A binary search has a runtime of $O(\log n)$. We start the search at the middle of the list, obtaining the value in the list at this point, and check whether or not this value is higher or lower than our target. Depending on our answer, we discard the half of the list in which our target is `not` located, and repeat the process, by obtaining the middle value of the half of the list we are currently searching, and repeating this until we reach our target. As we can tell, because the search space decreased by half for each iteration, i.e $(n, \frac{1}{2}n, \frac{1}{4}n, \frac{1}{8}n \cdots)$, we can conclude that this algorithm has a runtime of $O(\log n)$.

3 Slektstre

You shall make a software for genealogy where users can add persons and information who is a persons parents and who is a persons children (to keep this task simple we are not concerned with who is married to who and such, only biological parents/children). The problem you need to solve is: Given two persons, check if first person is a descendant of the second person.

```
public boolean isDescendant (Person person, Person ancestor) {

}
```

Explain what you need to do to solve this:

1. Explain which algorithm you would have used to implement the isDescendant method.
2. Explain which data structure you would use to store this tree of ancestors.
3. Give the running time of the algorithm. There are several possible choices of parameter to express the running time, n = ”number of persons in the tree” is perhaps the most natural, is there a more accurate measure to express the running time of the algorithm?

Svar. To store this ancestry tree we will use a directed graph, where each person is a node, and each node points to two parents (the persons biological parents), aswell as pointers to each of its biological children.

In order to implement the `isDescendant` method, we can use a Breadth-First Search. We the start-node of the BFS would be the ancestor, and in the `visited` hash set, we store all descendants that we have encountered so far. If the node we are currently searching is the person we are searching for, the method will return `true`, and if the person is not found, it will return `false`.

As far as describing the runtime of this algorithm, it will be in a worst case $O(n)$. This is because nodes are visited at most once, and there are n nodes in the graph. We could express the runtime by the amount of descendants a given ancestor has, as we would not bother searching any of the nodes which are not descendants of the current ancestor. This requires that we know how many descendants each person in the graph actually has, and this is not a given.

4 Heap

Explain how the data structure Heap functions. (In java it's called a `PriorityQueue`).

1. How is data stored in a heap?
2. What is the runtime to build a Heap containing `n` elements? Explain why. Is there a difference wheteher you insert elements one by one or insert all at once?

Svar. Data in a heap is stored as a binary tree. The data is stored in the binary tree in such a way that the elements with the smallest value will always be stored at the root, unless otherwise is specified. This means that whenever an element is removed from a heap, the next smallest value will now be stored at the root.

Inserting an element into an existing heap takes $O(\log n)$ time, and therefore, inserting n elements will take $O(n \log n)$ time. Inserting n elements at once into a heap, i.e. initalizing a `PriorityQueue` with an existing collection of elements, is a process often called `heapify`. This operation takes $O(n)$ time.

5 RandomList

You shall implement a simple list dat astructure which only needs two methods. You will receive point for:

- Correct implementation
- Effective implementation
- State and explain runtime for both methods.

Svar.

```java
import java.util.ArrayList;
import java.util.Collections;
import java.util.NoSuchElementException;
import java.util.Random;

public class RandomList<T> implements IRandomList<T>{
    private ArrayList<T> list;
    private Random rand = new Random();

    public void add (T elem){
        list.add(elem);
    }

    public T removeRandom(){
        if (list.isEmpty()) {
            throw new NoSuchElementException();
        }
        Collections.swap(list, rand.nextInt(list.size()), list.size() - 1);
        return list.remove(list.size() - 1);
    }
}
```

Given that we utlize an `ArrayList`, and that we are always adding to the tail of the list, the add method has a runtime of $O(1)$. Next, the `removeRandom` method swaps a random element with the element at the tail of the list, which runs in constant time. The return statement also runs in constant time, because we are removing from the tail of the list. This gives the final runtime for this method $O(1)$.

6 Central Value

In this task you shall implement a data structure.

1. Implement the interface `ICentralFinder.java`
2. Compute the runtime for each of the 3 methods.

Svar. The interface is implemented in the following manner

```java
import java.util.Collections;
import java.util.NoSuchElementException;
import java.util.PriorityQueue;

public class CentralFinder implements ICentralFinder{
    private PriorityQueue<Integer> hi = new PriorityQueue<>();
    private PriorityQueue<Integer> lo = new PriorityQueue<>(Collections.reverseOrder());

    @Override
    public void add(int number) {
        if (hi.isEmpty() || number > hi.peek()) {
            hi.add(number);
        } else {
            lo.add(number);
        }
        balance();
    }

    @Override
    public int removeCentralValue() {
        if (size() == 0) {
            throw new NoSuchElementException("List is empty");
        }

        balance();
        return hi.poll();
    }

    @Override
    public int size() {
        return lo.size() + hi.size();
    }

    private void balance() {
        if (hi.size() > lo.size() + 1) {
            lo.add(hi.poll());
        } else if (lo.size() > hi.size()) {
            hi.add(lo.poll());
        }
    }
}
```

The runtime of the method `balance` is determined by the following table

Operation	Runtime
.size()	$O(1)$
.add()	$O(\log n)$
.poll()	$O(\log n)$

This gives the final runtime for the `balance` method as $O(2\log n)$, or $O(\log n)$.

The `add` method has the runtime determined by the following table

Operation	Runtime
.isEmpty()	$O(1)$
.add()	$O(\log n)$
balance()	$O(\log n)$

This gives the final runtime of $O(2\log n)$ which is equal to $O(\log n)$.

The method `removeCentralValue` has the runtime determined by the following table

Operation	Runtime
.poll()	$O(\log n)$
balance()	$O(\log n)$

Which gives the final runtime of $O(2\log n)$ which is equal to $O(\log n)$.